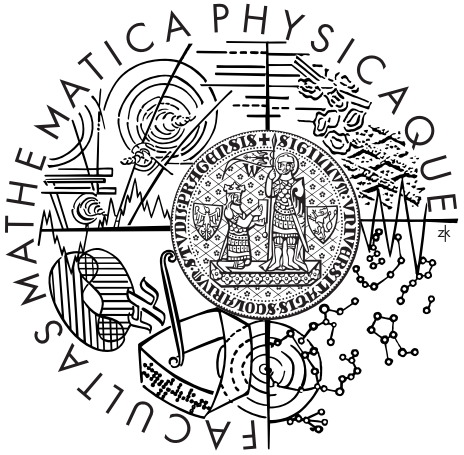
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Peter Zvirinský

Simulace mechaniky tuhých těles

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán
Studijní program: Informatika
Studijní obor: Programování

Praha 2011

Na tomto mieste by som sa chcel poďakovať svojmu vedúcemu práce RNDr. Josefovi Pelikánovi za úvod do problematiky, odborné rady a ochotný prístup. Taktiež by som sa rád poďakoval svojmu spolubývajúcemu RNDr. Branislavovi Dzurňákovi za cenné konzultácie a rady z oblasti fyziky.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 27. 5. 2011 Peter Zvirinský

Název práce: Simulace mechaniky tuhých těles

Autor: Peter Zvirinský

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán

Abstrakt: Predmetom tejto práce je vytvoriť rozšíriteľný fyzikálny simulátor tuhých telies. Súčasťou tohoto simulátoru bude detekcia kolízií v reálnom čase a následné spracovanie týchto kolízií. Simulátor bude schopný simulovať bežné sily vyskytujúce sa v prírode a ich vplyv na tuhé telesá. Medzi tieto sily patrí hlavne gravitácia, s ňou súvisiace rôzne druhy trenia a kľudové sily. Simulátor sa zameria predovšetkým na gule a na ich správanie sa vplyvom vyššie uvedených síl. Práca bude skúmať možnosti optimalizácie úloh ako detekcia kolízií a možnosťi paralelného prístupu.

Klíčová slova: 3D grafika, fyzikální simulace, tuhé těleso, tření, kolize

Title: Rigid body simulation

Author: Peter Zvirinský

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán

Abstract: The object of this work is to create an easily extendible rigid body simulation engine. The engine will contain real-time collision detection and realistic collision handling. It will be able to simulate common nature forces such as gravity, related different types of friction and resting forces. The simulation engine will mainly focus on spheres and their movement influenced by the forces mentioned before. The part of this work will be as well the examination of different types of optimalization of such tasks as collision detection and use of parallel approach.

Keywords: 3D graphics, physics simulation, rigid body, friction, collision

Obsah

1. Úvod

Fyzikálne založené modelovanie a simulácia je snaha preniesť prírodné javy a zákonitosti do sveta počítačového programu. Je založené na dvoch jednoduchých procesoch, ktorými sú matematické modelovanie a numerické výpočty. Matematické modelovanie sa zameriava na popis prírodných javov pomocou diferenciálnych rovníc, ktoré pokrývaju dynamiku a geometrickú reprezentáciu objektov. Numerické výpočty zas spočívajú v rýchlom a presnom hľadaní riešení týchto rovníc. Musia riešiť problémy ako numerické chyby vznikajúce kvôli konečnej presnosti reálnych čísel a obmedzenia výpočtovej sily.

1.1 Motivácia

V súčasnosti je fyzikálne založené modelovanie hojne využívanou metódou hlavne v oblastiach počítačovej grafiky, akými sú počítačové animácie, grafické modelovanie a predovšetkým počítačové hry. Aj napriek tomu, že začiatky moderných simulácii siahajú až do 80-tych rokov 20-teho storočia, je táto téma pre väčšinu záujemcov aj v súčasnosti príliš náročná a ťažko pochopiteľná. To je hlavne z dôvodu ťažko dostupných a príliš technických textov venujúcich sa tejto problematike, ktoré predpokladajú hlboké matematické a fyzikálne znalosti čitateľa. Vo svojej práci chcem priblížiť najbežnejšie používané metódy simulácie najjednoduchších objektov, ktorými sú tuhé telesá.

1.2 Cieľ

Pre vypracovanie tejto práce som si vymedzil niekoľko cieľov:

1. Preštudovať najčastejšie odporúčané zdroje venujúce sa tejto problematike, konkrétne fyzikálnemu modelovaniu tuhých telies.

2. Využiť získané informácie na návrh a implementáciu jednoducho rozšíriteľného a prenositeľného fyzikálneho enginu tuhých telies, riešiaceho všetky bežné vplyvy vyskytujúce sa v prírode, konkrétne gravitáciu, trenie a kľudové sily.

3. Zamerať sa predovšetkým na pohyb gúľ, ktoré poukazujú najlepšie na vplyv vyššie uvedených síl.

4. Pri implementácii využiť najčastejšie spôsoby optimalizácie a využiť možnosti paralelného prístupu.

5. Poskytnúť jednoduché demá, ktoré budu reprezentovať výsledok mojej práce.

2. Matematický model

2.1 Riešenie diferenciálnych rovníc

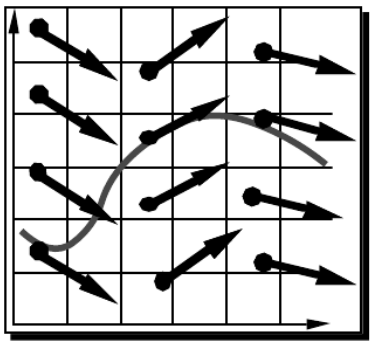
2.1.1 Obyčajné diferenciálne rovnice

Pre rozsah mojej bakalárskej práce si vystačím s touto podmnožinou diferenciálnych rovníc, ktoré pokryjú všetky požiadavky simulácie.

Obyčajne diferenciálne rovnice sa zvyčajne zapisujú vo forme

$$\dot{\mathbf{x}} = f(\mathbf{x}, t) \tag{2.1}$$

kde f je známa funkcia, $\mathbf{x}$ je stav systému a $\dot{\mathbf{x}}$ je derivácia $\mathbf{x}$ podľa času. V mojom prípade budú $\dot{\mathbf{x}}$ a $\mathbf{x}$ predstavovať vektory. Ďalej predpokladáme, že máme dané nejaké $\mathbf{x}(t_0) = \mathbf{x_0}$ v určitom čase t_0 a chceli by sme ďalej sledovať vývoj x v závislosti na čase. Tento problém je jednoduché si predstaviť v 2D, kde $x(t)$ predstavuje krivku popisujúcu pohyb bodu p v rovine. V ľubovolnom bode $\mathbf{x}$ môže funkcia f poskytnúť vektor, a teda f definuje určitú množinu vektorov v rovine, viď obrázok 2.1[1] nižšie. Vektor v určitom bode $\mathbf{x}$ predstavuje rýchlosť, ktorú musí mať bod p, ak bude niekedy prechádzať bodom $\mathbf{x}$.



Obr. 2.1: Derivácia funkcie $f(\mathbf{x}, t)$, predstavuje pole vektorov.

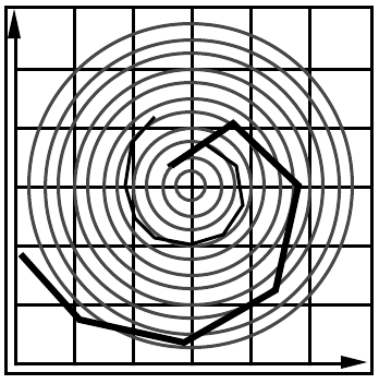
2.1.2 Numerické riešenie Eulerovou metódou

Štandardne sa tieto rovnice riešia symbolicky, ale to je časovo príliš náročné, a preto sa tento spôsob v real-time simuláciách nepoužíva. Eulerova metóda predstavuje najjednoduchšiu numerickú metódu riešenia týchto rovníc.

Definujme počiatočnú hodnotu pre naše $\mathbf{x}$ ako $\mathbf{x_0} = \mathbf{x}(t_0)$ a nech náš odhad pre x o nejaký časový krok neskôr $t_0 + h$ je $\mathbf{x}(t_0 + h)$, kde h predstavuje velkosť časového kroku. Eulerová metóda jednoducho spočíta $\mathbf{x}(t_0+h)$ tak, že spraví krok v smere derivácie v danom bode, ako

$$\mathbf{x}(t_0 + h) = \mathbf{x_0} + h\dot{\mathbf{x}}(t_0) \tag{2.2}$$

Táto metóda je veľmi jednoduchá ako na implementáciu tak na pochopenie, ale v niektorých prípadoch je veľmi nepresná (viď. obrázok 2.2[1]). Kvôli jednoduchosti našej simulácie, v ktorej na objekty bude pôsobiť (mimo nárazov riešených metódov impulzou, na ktorej presnosť nemá táto metóda priamy vplyv a je popísaná v sekcii 3.2.1) len gravitácia a trenie, táto metóda vystačí.



Obr. 2.2: Nepresnosť Eulerovej metódy zmení $\mathbf{x}(t)$ z kruhu na špirálu, tento efekt je možne zmierniť zmenšením kroku, nie však úplne odstrániť.

2.2 Dynamika častice

2.2.1 Charakteristika

Častica predstavuje najjednoduchší objekt simulácie, ktorý je reprezentovaný svojou hmotnosťou, pozíciou v priestore a rýchlosťou, kde prvá z menovaných vlastností je reprezentovaná skalárom a ostatné 3-vektorom. Ale ide o čiste abstraktný objekt, ktorý nemá žiadny tvar, čiže nezaberá žiadnu časť priestoru.

2.2.2 Translačný pohyb v priestore

Pohyb častice podľa Newtona je vyjadrený známym vzťahom $\mathbf{f} = m\mathbf{a}$ (Newtonov druhý zákon) alebo podľa značenia tohto textu ako $\ddot{\mathbf{x}} = \mathbf{f}/m$. Táto rovnica nie je v tvare popísanom v predošlej kapitole, obsahuje druhú deriváciu podľa času, čo z nej robí diferenciálnu rovnicu druhého rádu.

Túto rovnicu druhého rádu vyriešime spôsobom popísaným A. Witkinom a to tak, že ju prekonvertujeme na rovnicu prvého rádu zavedením novej premennej. Vytvoríme premennú v reprezentujúcu rýchlosť, čo nám dáva pár obyčajných diferenciálnych rovníc prvého rádu $\dot{\mathbf{v}} = \mathbf{f}/m$ a $\dot{\mathbf{x}} = \mathbf{v}$. Pozíciu a rýchlosť $\mathbf{x}$ a $\mathbf{v}$ takto môžeme spojiť do 6-vektoru. Teda pohyb je popísaný rovnicou $[\dot{x}_1, \dot{x}_2, \dot{x}_3, \dot{v}_1, \dot{v}_2, \dot{v}_3] = [v_1, v_2, v_3, f_1/m, f_2/m, f_3/m]$ a ak predpokladáme, že sila je funkcia $\mathbf{x}$ a t, tak dostávame rovnicu vo forme popísanej v predchádzajúcej kapitole a to $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$ [2].

2.3 Dynamika tuhého telesa

2.3.1 Charakteristika

Tuhé teleso je o niečo zložitejší objekt simulácie, pre ktorý platí všetko, čo aj pre časticu. Okrem toho však má tvar, a teda zaberá určitú časť priestoru, ktorá má nejaké geometrické vyjadrenie. Predpokladáme, že jeho tvar je pevne daný a nemenný. Kvôli tomu má tento objekt aj svoju orientáciu, ktorú je potrebné nejakým spôsobom reprezentovať, a taktiež s ňou súvisiacu rotačnú zložku pohybu.

2.3.2 Pozícia a orientácia

Pozícia objektu v priestore je vyjadrená klasicky 3-vektorom značeným $x(t)$, reprezentujúcim vzdialenosť jeho geometrického stredu[1] od počiatku súradnicovej sústavy. Orientácia objektu je vyjadrená rotačnou maticou 3×3, ktorú budeme značit $R(t)$. Matica $R(t)$ bude reprezentovať rotáciu objektu okolo jeho geometrického stredu ležiaceho v bode $(0, 0, 0)$ jeho lokálnej súradnicovej sústavy. Pozíciu ľubovoľného bodu p_0 tohto objektu v jeho lokálnej súradnicovej sústave transformujeme do svetových súradníc ako:

$$p(t) = R(t)p_0 + x(t) \tag{2.3}$$

Kedže $x(t)$ predstavuje pozíciu geometrického stredu v priestore, tak túto časť rovnice máme rovno danú.

2.3.3 Translačná a uhlová rýchlosť

Pre translačnú zložku pohybu platí všetko popísane v sekcii 2.2.2. Rotačnú zložku pohybu popíšeme uhlovou rýchlosťou, teda vektorom $\omega(t)$, kde jeho smer vyjadruje os a jeho velkosť $|\omega(t)|$ rýchlosť otáčania. Zmena rotácie $\dot{R}(t)$ je potom podľa [3] vyjadrená ako:

$$\dot{R}(t) = \omega(t)^{*^2} R(t) \tag{2.4}$$

Rýchlosť ľubovolného bodu daného tuhého telesa dostaneme ako [3]:

$$\dot{r}(t) = \omega(t) \times (r(t) - x(t)) + v(t) \tag{2.5}$$

kde r značí pozíciu bodu v svetových súradniciach a $\dot{r}$ rýchlosť daného bodu.

Ďalej označme $F_i(t)$ celkovú silu pôsobiacu na i-tú časticu telesa v čase t. Potom definujeme krútiaci moment pôsobiaci na i-tú časticu ako [3]:

$$\tau_i = (r_i(t) - x(t)) \times F_i(t) \tag{2.6}$$

Celkovú silu pôsobiacu na teleso v čase t potom dostaneme ako[3]:

$$F(t) = \sum F_i(t) \tag{2.7}$$

A celkový krútiaci moment pôsobiaci na teleso v čase t podobne [3]:

$$\tau(t) = \sum \tau_i(t) \tag{2.8}$$

[1] Toto nemusí platiť všeobecne, ale pre rozsah telies pokrytých touto prácou s touto myšlienkou vystačíme.

[2] Notáciu * definujeme ako $a^*b = \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix} \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = a \times b$

3. Kolízie

3.1 Detekcia kolízií

3.1.1 Základný algoritmus

Dva objekty považujeme za kolidujúce, ak ich prienik je neprázdna množina bodov[1]. Nájsť všetky kolidujúce objekty v scéne je sám o sebe problém $O(N^2)$ časovej zložitosti, kde N značí počet objektov, pretože je potrebné prejsť všetky dvojice predmetov a otestovať ich na kolíziu. Ale keďže každú dvojicu stačí prejsť raz, pretože kolidovať predstavuje symetrickú reláciu, dostávame časovú zložitosť $O(N^2/2)$.

Úlohou samotného algoritmu však nie je len nájsť dvojice penetrujúcich objektov, ale aj presný čas ich kolízie v rámci daného časového úseku.

Algoritmus prejde všetky dvojice objektov pred každou aktualizáciou polôh objektov, teda na začiatku kroku simulácie. Na vstup dostane veľkosť časového úseku, o ktorý sa chce simulácia posunúť dopredu. Následne každú dvojicu objektov posunie na koniec časového úseku a otestuje ich na prienik. Ak prienik nenastal, znamená to, že objekty v danom časovom kroku určite nekolidujú. V opačnom prípade prienik nastal a je potrebné vrátit objekty späť na začiatok časového kroku, kedy objekty ešte nekolidovali. Potom algoritmus prejde znovu celý časový úsek s väčšou jemnosťou a nájde presný čas kolízie.

Takto spracuje všetky dvojice objektov a vráti získané časy simulácii, ktorá musí kolízie správne spracovať a posunúť sa o vhodný časový úsek dopredu.

Je dôležité myslieť na to, že algoritmus počíta s tým, že na začiatku časového úseku daná dvojica určite nekoliduje, túto podmienku musí zaručiť samotná simulácia spôsobom akým sa posúva dopredu. Spracovávanie kolízií preto prebieha tak, že vždy sa nájde najbližšia kolízia, do ktorej času sa simulácia následne posunie. Kolízia je potom ošetrená a celé testovanie je potrebné zopakovať znovu, pretože zmeny spôsobené touto kolíziou mohli vyvolať kolízie nové.

Algoritmus taktiež nepočíta s veľmi rýchlymi objektami, pretože ak sa stane, že časový krok je vzhľadom k rýchlosti dvojice objektov príliš veľký a na konci časového úseku je ich kolízia vyhodnotená záporne, tak táto kolízia nebude detekovaná. Pre účely simulácie v rozsahu tejto práce však posledne menovaný problém nie je potrebné riešiť, pretože všetky objekty budú mať relatívne malú rýchlosť vzhľadom k svojej veľkosti.

3.1.2 Optimalizácie

Takto popísaný algoritmus je veľmi jednoduchý na implementáciu aj na pochopenie, je však veľmi nepraktický, pretože robí vela práce zbytočne. Síce nie je možné znížiť jeho časovú zložitosť, ale k dispozícii je niekoľko bežne využívaných optimalizácii, ktoré sa hodia do simulácie rozsahu tejto prace a tie popíšem v tejto kapitole.

[1] Popis geometrického riešenia problému prieniku objektov pokrytých touto prácou je možné nájst v [4].

Filtrovanie dvojíc

V každej simulácii je možné rozdeliť objekty na dva typy, a to pohyblivé a nepohyblivé, resp. dynamické a statické. Vo veľkom množstve príkladov, akými sú napríklad počítačové hry, presahuje množstvo statických objektov množstvo tých dynamických. Je preto vhodné rozdeliť si objekty na tieto dve skupiny a zamedziť nezmyselnému porovnávaniu dvojíc statických objektov, ktoré spolu nemôžu nikdy kolidovať.

Segregácia

Pri pohľade na rozsiahlu scénu je na prvý pohľad jasné, že napríklad dva objekty nachádzajúce sa na protiľahlých koncoch scény spolu nemôžu kolidovať. Preto je veľmi výhodné rozdeliť si scénu na časti a spracovávať kolízie len v rámci jednej časti.

Tu je dôležité si uvedomiť, že drvivá väčšina scén je orientovaná vzhľadom k nejakej rovine, na ktorej sa nachádzajú ďalšie objekty a má vzhľadom k svojej šírke a dĺžke relatívne malú výšku. Preto stačí rozdeliť scénu na časti vzhľadom k tejto rovine.

Pri výbere spôsobu reprezentácie je treba dbať na to, že scéna je pohyblivá a je potrebné ju aktualizovať po každom kroku simulácie, čo kladie veľké nároky na rýchlosť vkladania prvkov.

Najčastejšie využívanou reprezentáciou je takzvaná 2D-mriežka[2], ktorá rozdelí svet na rovnako veľké bunky a v každom kroku jednoducho spočíta bunky, v ktorých sa daný objekt nachádza. Tento výpočet sa na začatku urobí pre všetky statické objekty a v ďalších iteráciách sa aktualizujú už len dynamické objekty.

Paralelizácia

Detekcia kolízií je veľmi jednoducho paralelizovateľný proces, pretože nič nebráni tomu, aby sa rozličné dvojice objektov testovali na kolíziu súčasne. Samozrejme je treba sa vyvarovať prípadov, kedy dvojice spracovávané súčasne obsahujú rovnaký objekt, ale tento jav je možné vyriešiť jednoduchým skopírovaním objektu alebo synchronizáciou na úrovni prístupu.

3.2 Reakcia na kolíziu

V tejto kapitole bude popísaný jeden z bežne využívaných spôsobov riešenia kolízií medzi tuhými objektami a to metódou impulzov. Celá kapitola sa bude zameriavať len na kolízie, kde je kontakt medzi kolidujúcimi objektami tvorený len jedným bodom[3].

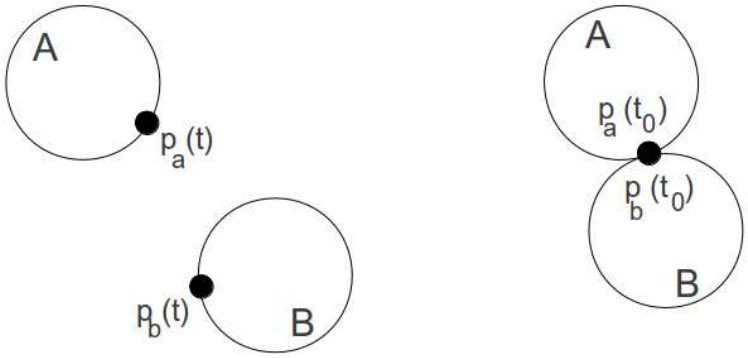
3.2.1 Impulz

Predpokladajme, že máme danú dvojicu objektov A, B a bod ich kolízie p. Označme $p_a(t)$ bod objektu A, ktorý spĺňuje $p_a(t_0) = p$, kde t_0 predstavuje čas kolízie

[2]Bližší popis a informácie o výhodách je možné nájsť v [5] a [6]

[3]Zložitejšie typy kolízii, kde je kontakt tvorený niekoľkými bodmi alebo nejakou plochou, je mimo rámec tejto práce a čiastočný popis ich riešenia je možne nájsť v [7].

a podobne $p_b(t)$ bod objektu B splňujúci $p_b(t_0) = p$. Aj keď sú body p_a, p_b v čase kolízie totožné, ich rýchlosti určite nemusia byť. Pre správne ošetrenie kolízie je potrebné preskúmať rýchlosti týchto dvoch bodov v čase kolízie.



Obr. 3.1: Body p_a, p_b pred a počas kolízie

Podľa rovnice (2.5) je možné zistiť rýchlosť daného bodu $\dot{p_a}(t_0)$ ako:

$$\dot{p_a}(t_0) = v_a(t_0) + \omega_a(t_0) \times (p_a(t_0) - x_a(t_0)) \tag{3.1}$$

Kde $v_a(t)$ a $\omega_a(t)$ sú rýchlosti telesa A. Úplne rovnakým spôsobom len z odlišnými rýchlosťami je vyjadrená rýchlosť bodu $p_b(t_0)$ telesa B. Najdôležitejšou veličinou pre výpočet odrazu je relatívna rýchlosť pôsobiaca kolmo vzhľadom k rovine nárazu a je vyjadrená ako [7]:

$$v_{rel} = \hat{n}(t_0) \cdot {}^{4}(\dot{p_a}(t_0) - \dot{p_b}(t_0)) \tag{3.2}$$

kde $\hat{n}$[5](t_0) predstavuje jednotkový vektor kolmý na rovinu nárazu.
Z hodnoty v_{rel} je potom možné rozdeliť kolízie na tri typy a to[7]:

- $v_{rel} < 0$ - dochádza k zrážke telies a v prípade neošetrenia tejto kolízie dôjde k prieniku telies.
- $v_{rel} > 0$ - telesá sa od seba vzďaľujú, pravdepodobne preto, že kolízia bola už spracovaná v predchádzajúcom kroku simulácie a nie je potrebné sa jej ďalej venovať.
- $v_{rel} = 0$ - telesá sa dotýkaju a sú v kľude[6].

V tejto časti textu nás zaujíma prvý prípad, pretože je potrebné nejakým spôsobom zmeniť rýchlosti objektov, aby sa zabránilo ich prieniku. Na ošetrenie tejto požiadavky zavedieme novú veličinu J, ktorú nazveme impulz [7]. Bude to vektorová veličina veľmi podobná sile pôsobiacej na teleso. Jediný rozdiel je v tom, že impulz mení rýchlosť objektu okamžite na rozdiel od bežnej sily, ktorá pôsobí

[4] $\cdot$ značí skalárny súčin.

[5] Bližší popis a príklady k definícii tejto normály je možné nájsť v [7] a [4], vzhľadom k obmedzenému rozsahu telies pokrytých touto prácou je spôsob výpočtu tejto normály nechaný na čitateľa, pretože ide o veľmi triviálny proces.

[6] Tomuto typu kolízií je venovaná celá nasledujúca sekcia.

v čase. Na to, aby sme zistili presné pôsobenie impulzu J, si predstavíme veľmi veľkú silu F, ktorá pôsobí vo veľmi krátkom časovom intervale Δt. Ak necháme F ísť limitne do nekonečna a Δt do nuly tak, že [7]

$$F\Delta t = J \tag{3.3}$$

uvážením zmeny rýchlosti telesa pôsobením F počas časového intervalu Δt. Veľkosť impulzu, ktorú označíme j, potom dostaneme ako [7]

$$j = \frac{-(1+\epsilon)v_{rel}^{-}}{\frac{1}{M_a} + \frac{1}{M_b} + \hat{n}(t_0) \cdot (I_a^{-1}(t_0)(r_a \times \hat{n}(t_0))) \times r_a + \hat{n}(t_0) \cdot (I_b^{-1}(t_0)(r_b \times \hat{n}(t_0))) \times r_b} \tag{3.4}$$

kde

- M_a resp. M_b predstavuje hmotnosť telesa A resp. B
- I_a^{-1} resp. I_b^{-1} predstavuje inverznú maticu tenzoru zotrvačnosti telesa A resp. B
- r_a resp. r_b je vektor smerujúci zo stredu telesa A resp. B do bodu kolízie
- v_{rel}^{-} je relativná rýchlosť v smere $\hat{n}(t_0)$, tesne pred kolíziou
- ϵ je koeficient pružnosti

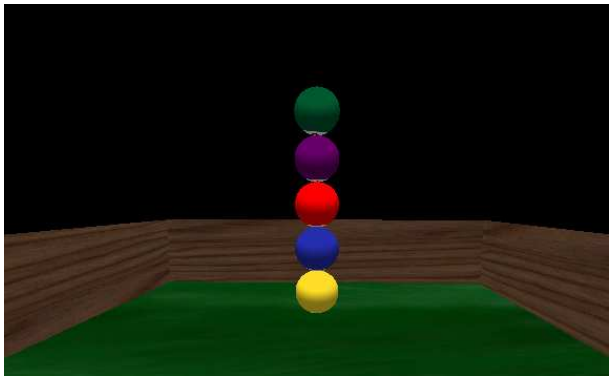
Tento impulz je v prípade kolízie bez trenia aplikovaný v smere $\hat{n}(t_0)$ resp. v protismere pre teleso A resp. B.

3.2.2 Kľudové sily

V tejto sekcii sa budem venovať kolíziám, pre ktoré platí ${v_{rel}}^{7} = 0$, presnejšie je blízko nuly, táto minimálna medz je nastavená podľa potreby scény, väčšinou podľa veľkosti gravitačnej sily. Toto je potrebné riešiť oddelene kvôli tomu, aby sa simulácia zbytočne nezaťažovala veľmi malými a častými kolíziami. Pre príklad uveďme guľu pustenú z určitej výšky na podložku (predpokladáme, že jej koeficient pružnosti je menši ako 1), po niekoľkých kolíziách by bola sila, ktorou by guľa narážala o podložku nesmierne malá, a tak isto by bol aj čoraz menší jej odraz od podložky. Tento náraz by sa zmenšoval, až by narazil na obmedzenú presnosť reálnych čisel a veľmi pravdepodobne by guľa prepadla pod podložku. Miera kolízií za časový úsek by sa taktiež veľmi rýchlo zvyšovala, až na niekoľko desiatok kolízií pre jediný objekt v scéne za jeden časový krok simulácie.

Úlohou tejto sekcie je navrhnúť riešenie výpočtu kľudových síl, ktoré by zabránilo prieniku telies, ale nevytváralo nežiaduce efekty ako neprirodzené skackanie objektov po podložke a pod. Tieto kľudové sily sa musia spočítať naraz pre všetky objekty, pretože jedno teleso môže byť v kontakte aj s niekoľkými objektami, viď obrázok nižsie. Taktiež je potrebné, aby sa tieto pokojové sily spočítali len raz na začiatku každého kroku simulácie, a tým nezaťažovali jej beh.

[7] Pre definíciu viď predchádzajúcu sekciu.



Obr. 3.2: Efekt, ktorý je potrebné dosiahnuť. Predstavuje niekoľko na seba položených objektov, ktoré zostávaju v pokoji bez akýchkoľvek vedľajších efektov.

Formulácia problému

Majme danú nejakú množinu kontaktov, ktoré je potrebné ošetriť, každý kontakt označme p_i a skalárom a_i označme relatívne zrýchlenie pozdĺž normály[8] nárazu medzi dvoma objektami tvoriacich tento kontakt. Aby sa zabránilo prieniku, je potrebné udržať $a_i \geq 0$ pre všetky kontaktné body. Ďalej označme skalárom f_i silu pôsobiacu medzi dvoma telesami pozdĺž normály kolízie. Pre odpudivú silu bude platiť $f_i \geq 0$ a pre príťažlivú opak. Keďže chceme, aby boli všetky sily odpudivé, potrebujeme dodržať taktiež podmienku $f_i \geq 0$. Kvôli zachovaniu energie (stále uvažujeme kontakt bez trenia) je potrebné dodržať ešte podmienku $f_i a_i = 0$. Toto nám zaručí, že ak je $a_i = 0$, tak kontakt zotrvá a ak $a_i \geq 0$, tak sa kontakt zruší a $f_i = 0$. Ak označíme vektorom $\mathbf{f}$ sériu všetkých f_i a vektorom a sériu odpovedajúcich a_i, potom môžme ich vzťah vyjadriť rovnicou v tvare [7]

$$\mathbf{a} = \mathbf{A}\mathbf{f} + \mathbf{b} \tag{3.5}$$

kde $\mathbf{A} \in \mathbb{R}^{n \times n}$ je symetrická a pozitívne semidefinitná matica a $\mathbf{b} \in \mathbb{R}^n$ je vektor stĺpcového priestoru $\mathbf{A}$[9].

Pomocou tohto zápisu môžme požadované podmienky vyjadriť ako [7]

$$\mathbf{A}\mathbf{f} + \mathbf{b} \geq \mathbf{0},\ \mathbf{f} \geq \mathbf{0},\ \ \mathbf{f}^T(\mathbf{A}\mathbf{f} + \mathbf{b} = \mathbf{0}) \tag{3.6}$$

Algoritmus

Rovnice (3.6) zo sekcie vyššie vytvárajú takzvaný problém lineárnej komplementarity a jedno z riešení takéhoto problému spočíva v použití kvadratického programovania [7]. Problém tohto riešenia je malé množstvo kvalitných knižníc podporujúcich riešenie úloh kvadratického programovania. Implementácia takejto knižnice by ďaleko presahovala rozsah tejto práce, a preto poskytnem popis alternatívneho algoritmu využívajúceho len riešenie bežnej sústavy rovníc. Takéto knižnice sú bežne dostupné a jednoduché na použitie. Presný popis a rozbor algoritmu je potom možné nájsť v [8].

Samotný algoritmus pracuje nasledovne - na začiatku sú všetky kontaktné body ignorované a všetky f_i sú nastavené na 0. Algoritmus začne spočítaním

[8] Bližší popis je možné nájsť v prechádzajúcej sekcii.

[9] Presnejší význam tejto matice a návod na jej získanie je možné nájsť v [7]

hodnoty f_1 tak, aby splnil podmienky pre $i = 1$. Následne spočíta f_2 tak, aby splňovalo podmienky pre $i = 2$, zatiaľčo dodrží podmienky pre $i = 1$. To kľudne môže vyžadovať modifikáciu f_1. Takto algoritmus pokračuje ďalej a v ľubovoľnonom bode algoritmu platí, že pre nejaké k a kontaktné body $1 \leq i \leq k-1$ sú podmienky dodržané a pre $i > k$ platí $f_i = 0$ a algoritmus v tomto kroku spracuje f_k tak, aby dodržal všetky podmienky pre $i \leq k$, aj s možnosťou ich modifikácie. Algoritmus končí, ak sú podmienky splnené pre všetky body.

3.2.3 Trenie

Šmykové trenie

Riešenie trenia popísane v tejto sekcii je čiastočne inšpirované myšlienkami zhrnutými v [9].

Doteraz sme sa spoliehali, že v kontaktnom bode nenastáva žiadne trenie a impulz, ktorý bol výsledkom nárazu, bol aplikovaný vždy v smere normály roviny kolízie. Trenie však pôsobí proti tangencialnej (rovnobežnej[10]) zložke rýchlosti v čase nárazu, a tým vychýli smer aplikácie impulzu.

Vplyv trenia a veľkosť tangenciálneho impulzu značeného ako p_{T_i} v mieste kolízie je daný ako

$$p_T = \mu_s j_N \tag{3.7}$$

kde j_N značí veľkosť normálového impulzu j zo sekcie 3.2.1 a μ_s je koeficient statického trenia. Rovnako je daný vplyv dynamického trenia len s rozdielnou konštantou μ_d.

Prvý naivný pokus odporúčaný a popísaný v [10] tvrdí, že stačí aplikovať tento impulz proti smeru tangencialnej zložky rýchlosti v bode nárazu a nie je potrebné rozdeľovať trenie na dynamické a statické. Tento zjednodušený prístup má však jednu veľkú medzeru a to, že môže do sústavy pridávať energiu. Ako príklad uvediem telesá, ktorých relatívna tangenciálna zložka rýchlosti je menšia než p_T. Aplikácia trenia v tomto prípade vyvolá otočenie tangenciálnej rýchlosti telies a pridá doň energiu, ktorá ma za následok veľmi neprirodzené správanie sa.

Je dôležité si uvedomiť, že v skutočnosti je vplyv trenia vyjadrený skôr nerovnicou

$$p_T \leq \mu_x j_N \tag{3.8}$$

pre oba druhy trenia.

Na základe tohto zistenia je fyzikálne správne aplikovať trenie nasledovným spôsobom. Impulz je definovaný v [7] ako sila potrebná na úplné zastavenie telies, čo je vhodné využiť. V tomto prípade sa vypočíta tangenciálny impulz značený j_T, na ktorého výpočet je možne aplikovať vzorec (3.4) z predošlej sekcie. Jediný rozdiel bude v zmene vektoru pôsobenia sily z normály na vektor rovnobežný s rovinou nárazu. Následne ak platí $\mu_s j_N < j_T$, tak sa na telesá aplikuje impulz j_T, v opačnom prípade impulz $\mu_d j_N$. Myšlienka za tým je zjavná, ak je tangencálna zložka sily pôsobiaca na telesá menšia ako $\mu_s j_N$, čo je veľkosť statického trenia, tak to znamená, že statické trenie nebolo prekonané a je nutné znížiť relatívnu tangenciálnu rýchlosť na nulu, v opačnom prípade bolo statické trenie prekonané a aplikuje sa trenie dynamické.

[10] Pretože je rovnobežná s rovinou nárazu.

Valivé trenie

V tejto práci som sa rozhodol zamerať na gule, a preto je potrebné rozmýšlať ešte o jednom druhu trení a to valivom. Válivé trenie značené ako F_r je druh trenia, ktoré pôsobí medzi telesom kruhového prierezu a podložkou. Toto trenie začne pôsobiť ak relatívna rýchlosť v bode kontaktu dosiahne 0 [11].

Jeho veľkosť je definovaná ako [11]

$$F_r = \frac{\mu_r F_N}{r} \tag{3.9}$$

kde μ_r predstavuje koeficient valivého trenia a F_N je sila, ktorou tlačí teleso na podložku. Toto trenie pôsobí na stred telesa a proti smeru pohybu.

4. Implementácia

Táto kapitola bude obsahovať stanovenie presných cieľov pre softwarovú časť práce a odôvodnenie výberu platformy. Nasledovať bude bližší popis implementácií niektorých dôležitých častí. Tento popis bude obsahovať len hlavnú myšlienku, kompletný popis tried a jej členských metód je súčasťou programátorskej dokumentácie v dodatku C. Kapitola bude zakončená popisom jednotlivých dém, ktoré sú súčastou práce.

4.1 Ciele implementácie

Cieľom projektu je návrh a implementácia ľahko rozšíriteľného a prenositeľného fyzikálneho enginu, ktorý bude pokrývať bežné sily pôsobiace na tuhé teleso v prírode. Tieto sily sú konkrétne gravitácia a s ňou súvisiace trenie. Simulácia sa zameria hlavne na gule, pretože ich pohyb najlepšie demonštruje vplyv vyššie zmienených síl. Veľký dôraz bude kladený na objektový model aplikácie a možnosť jej jednoduchého rozšírenia. Celá aplikácia bude navrhnutá tak, aby jej prezentačná vrstva bola oddelená od tej výpočtovej, a tak umožní jej jednoduchú výmenu v prípade potreby. Súčasťou aplikácie budú demá, ktoré budú demonštrovať fungovanie tohto enginu.

Veľké úsile pri implementácii bude vydané na možné optimalizácie a taktiež využitie paralelného prístupu.

Tento systém bude prenositeľný na operačné systémy Linux a Windows.

4.2 Analýza

Najdôležitejšie vlastnosti vyžadované od aplikácie sú prenositeľnosť a rýchlosť. Vzhľadom k tomu, že aplikácia používa výpočty v reálnom čase, je nevhodné použiť nejaký vysoko úrovňový programovací jazyk s automatickou správou pamäti, pri kombinácii s prenositeľnosťou a dostupnosťou knižníc ako jediná možnosť ostáva jazyk C alebo C++. Veľkou výhodou v tomto prípade je takmer úplná absencia GUI a s tým súvisiacich problémov. Kvôli objektovým nárokom aplikácie a jej jednoduchej rozšíriteľnosti bol nakoniec zvolený jazyk C++.

Ako prezentačná vrstva bola znovu kvôli prenositeľnosti zvolená knižnica OpenGL.

4.3 Implementačne poznámky

4.3.1 Simulácia

Algoritmus hľadania kolízií je presne popísaný v sekcii 3.1.1. Táto poznámka sa týka jednej jeho časti, a to hľadania presného času kolízie. Jeden jednoduchý spôsob ako ho nájsť je znovu lineárne prejsť časový interval menším krokom, v ktorom kolízia nastala. Tento problém však je možné riešiť aj klasickým binárnym vyhľadávaním pre reálne čísla, čo rapídne zníži počet potrebných posunov objektu v rámci časového intervalu.

Tento spôsob však so sebou nesie niekoľko problémov, a to hlavne nestabilitu. Tá sa prejavuje zacyklením a nastáva hlavne, ak sa hľadaná kolízia nachádza veľmi blízko začiatku alebo konca časového úseku. Pri implementácii bol použitý jednoduchý spôsob prevencie zacyklenia, a to obmedzenie počtu iterácií. Ten pracuje nasledovne - ak hlavný cyklus vyhľadávania prekročí maximálny počet krokov bez toho aby našiel čas simulácie, zoberie sa pôvodná kolízia, ktorá bola detekovaná pred vstupom do vyhľadávania a priradí sa jej aktuálny čas, v ktorom nastalo prekročenie maximálneho počtu krokov. Hlavný cyklus vyhľadávania pre statické[1] objekty potom vyzerá nasledovne:

```
// inicializácia riadiacich premenných
double start = 0.0;
double end = time_step;
double mid;
// BIN_SEARCH_ACCURACY je definovaná konštanta, ktorá určuje presnosť
// vyhľadávania
double fine_step = time_step / BIN_SEARCH_ACCURACY;

\\ počítač počtu krokov cyklu
int counter = 0;
Collision * col = NULL;
while (end - start >= fine_step || !col)
{
        if (col)
                delete col;
        // Detekovanie zacyklenia a˜vyššie popísane ošetrenie
        if (counter++ > MAX_STEPS_BIN_SEARCH)
        {
                col_init->time = mid;
                dynamic_entity->reverse();
                return col_init;
        }

        mid = (start + end) / 2;
        dynamic_entity->onUpdate(mid);
        if (!(col = collision_detector.detect(dynamic_entity, entity)))
        {
                start = mid + fine_step;
        }
        else
        {
                end = mid - fine_step;
        }
        // Reset the object.
        dynamic_entity->reverse();
}
```

[1] Kvôli optimalizácii má tento algoritmus 2 verzie, jedna pre statické a druhá pre dynamické objekty.

4.3.2 Paralelizácia

Paralelizácia programu prebieha na dvoch úrovniach, prvá úroveň umožňuje beh prezentačnej vrstvy paralelne s výpočtami simulácie a druhá paralelizuje proces detekcie kolízií.

Prvá úroveň

Prvá spomínaná úroveň je veľmi jednoduchá, pretože je potrebné synchronizovať prístup len k jednému objektu a to kontajneru[2] všetkých telies simulácie. Táto synchronizácia je uskutočnená cez zámok prístupný od simulácie metódou `getLock()`. Kvôli vyhnutiu sa aktívnemu čakaniu je k dispozícii aj podmienená premenná cez metódu `getCondition()`, pomocou ktorej je možné uspávať a budiť jednotlivé vlákna. Predtým než prezentačná vrstva zažiada od simulácie objekty určené na vykreslenie, je povinná najprv zamknúť si vyššie zmienený zámok a po ich získaní ho okamžite uvoľniť. Tento zámok používa aj simulácia, tá si ho zamkne pred začatím výpočtov a uvoľní po ich skončení. Simulácia sa po každom kroku uspí a čaká na vyššie zmienej podmienenej premennej, kým nedostane notifikáciu o tom, že sa má znovu posunúť o ďalší krok. Tento prístup umožňuje prezentačnej vrstve kontrolovať fps[3] a tým aj rýchlosť simulácie.

Druhá úroveň

Táto úroveň je náročnejšia na prevedenie a je reprezentovaná synchronizovanou centrálnou jednotkou[4] obsahujúcou zoznam všetkých dvojíc, ktoré je potrebné otestovať na kolíziu. Má za úlohu taktiež spravovať vlákna[5] spracuvávajúce tieto dvojice, zabezpečíť zbieranie ich dát a spracovať ich do tvaru, v ktorom sú následne predané simulácii. Počet vlákien je v tomto prípade premenný, čo umožňuje využiť maximálny potenciál procesoru pre túto úlohu. Tieto vlákna sú vytvorené spolu so simuláciou a existujú počas celého jej behu. Pri ich vytváraní je im predaný odkaz na túto centrálnu jednotku, z ktorej získavajú dvojice na spracovanie. Každé toto vlákno sa uspí hneď potom ako sa zoznam dvojíc pre spracovanie vyprázdni a ony skončia svoje výpočty. Je znovu úlohou riadiacej jednotky zobudiť tieto vlákna pri jej opätovnom naplnení.

4.4 Demonstračné príklady

K dispozícii je spolu 7 dém, ktoré prezentujú jednotlivé vlastnosti enginu. Popis spustenia aplikácii a jednotlivých dém je súčasťou užívateľskej dokumentácie prílohy B.

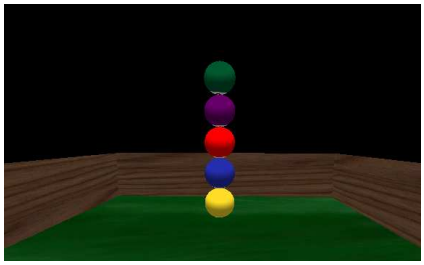
[2] V dodatku C viď EntitiesContainer
[3] Frames per second - počet prekreslení scény za sekundu.
[4] V dodatku C viď CollisionDetectionWorkCollector
[5] V dodatku D viď DetectionCheckWorker

demo č. 0

Je tvorené stolom s biliardovým rozložením gúľ a obsahuje jednoduché GUI, pomocou ktorého je možné výskušať si správanie enginu.

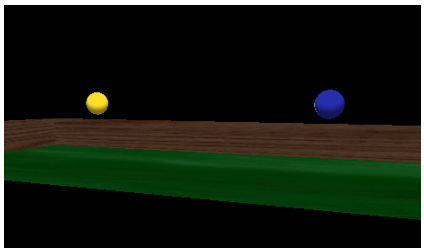
demo č. 1

Niekoľko na seba položených gúľ, ktoré demonštrujú výsledok algoritmu pre počítanie kľudových síl. Nie je možné vidieť žiadne neprirodzené vibrovanie ani iný vedľajší efekt.



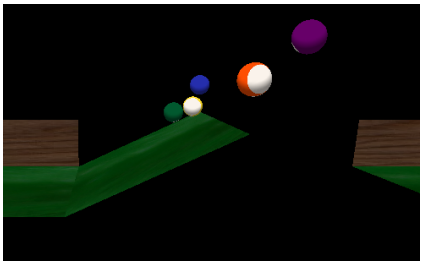
demo č. 2

Toto demo znázorňuje vplyv trenia na odraz gule od podložky. Simulacia obsahuje 2 gule padajúce z veľkej výšky, pričom prvá z nich má nejakú nenulovú uhlovú rýchlosť a druhá nulovú.



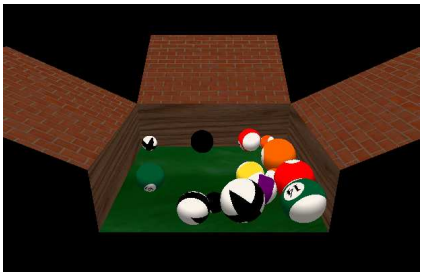
demo č. 3

Znázorňuje vplyv hmotnosti na pohyb gule. Scéna je tvorená dvoma oddelenými stolmi prepojenými skokanským mostíkom. Každá guľa potrebuje dostať určitú rýchlosť na to, aby preletela z jedného stolu na druhý.



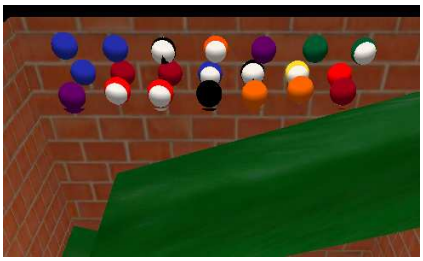
demo č. 4

Demonštruje nárazy rôzne veľkých gúľ s rôznymi hmotnosťami, ktoré sú pustené z rôznych výšok a sú nasmerované tak, aby sa stretli v strede stola.



demo č. 5

Demonštruje vplyv gravitácie na gule. Scéna pozostáva z väčšieho množstvá gúľ, ktoré sú z výšky pustené na nerovnú podložku a vplyvom gravitácie sa skotúľaju až na spodok scény.



demo č. 6

Toto demo slúži na testovacie účely, je možné v ňom vygenerovať ľubovoľné množstvo gúľ, a tak testovať výkon enginu.

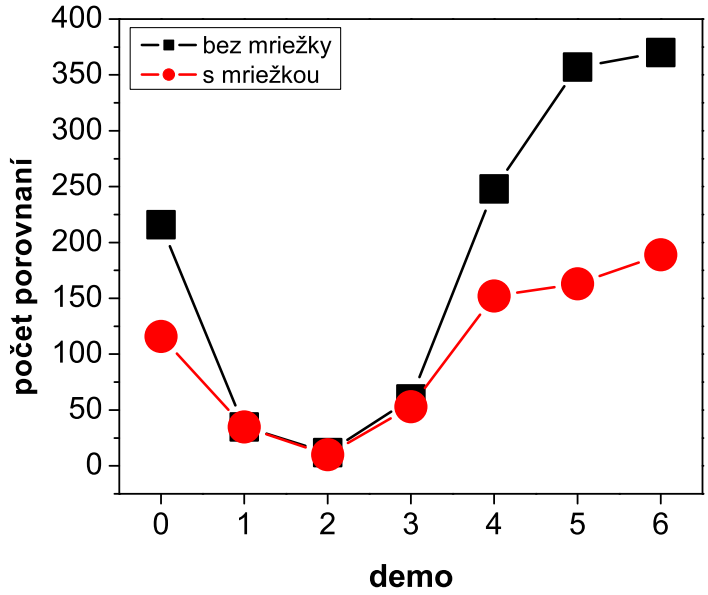
5. Zhrnutie

5.1 Výsledky

Výsledkom práce sa stal fyzikálny engine s pomerne komplexným objektovým modelom, ktorý však poskytuje možnosť jednoduchého rozšírenia a univerzálneho prístupu. Podľa pôvodného plánu je prezentačná vrstva oddelená od simulačnej, čo značne zvyšuje využiteľnosť enginu.

Simulácia nárazov zvoleným spôsobom vykazuje veľmi vierohodné výsledky a taktiež sa do tohoto modelu podarilo vhodným spôsobom zapracovať vplyv trenia, na základe ktorého je pohyb gúľ akoby základného telesa veľmi prirodzený.

Mriežka ako zvolená urýchľovacia dátová štruktúra znižuje počet porovnaní na priložených demách priemerne o 31%. Pre presné namerané hodnoty viď graf nižšie.



Obr. 5.1: Vývoj počtu porovnaní pre jednotlivé demá.

Podarilo sa vo veľkej miere využiť paralelizmus, ktorého účinnosť bola meraná na dvoch počítačoch s rôznymi procesormi. Prvý z nich predstavuje bežne dostupnú alternatívu s dvojjadrovým procesorom, a kvôli možnosti otestovať paralelizáciu s väčším počtom vlákien je druhou alternatívou počítač s osemjadrovým procesorom. Poskytnuté sú len hodnoty pre demá č. 3-6, pretože ostatné sú príliš jednoduché a ich namerané hodnoty sa nachádzajú na hranici 1 ms. Časy obsiahnuté v tabuľkách a grafoch predstavujú priemer trvania jedného kroku simulácie. Meranie prebiehalo vždy od začiatku simulácie až do úplneho zastavenia gúľ. Namerané hodnoty pre jednotlivé úrovne simulácie[1] je možné vidieť nižšie.

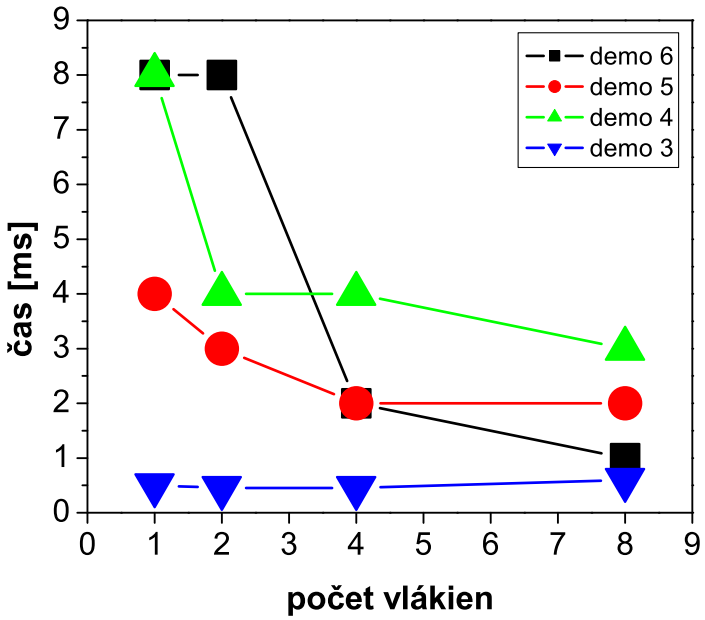
[1]Paralelizácia 1. úrovne znamená oddelenie behu prezentačnej vrstvy od výpočtov simulácie a 2. úroveň paralelné spracovávanie detekcie kolízií. Pre detail viď kapitolu 4.

Demo č.	Bez paralelizácie	Paralelizácia 1. úrovne
3	15	10
4	36	19
5	31	20
6	39	26

Tabuľka 5.1: Namerané hodnoty (v milisekundách) pre prvý testovací počítač s dvojjadrovým procesorom Intel Core 2 Duo (2.10 GHz).

Demo č.	Bez paralelizácie	1. úroveň	2. úroveň - 2 vlákna	2. úroveň - 4 vlákna	2. úroveň - 8 vlákien
3	4	0.5	0.45	0.45	0.6
4	23	8	4	4	3
5	20	4	3	2	2
6	23	8	8	2	1

Tabuľka 5.2: Namerané hodnoty (v milisekundách) pre druhý testovací počítač s osemjadrovým procesorom Intel Core I7 (2.67GHz).



Obr. 5.2: Graf závislosti času trvania jedného kroku simulácie vzhľadom k rastúcemu počtu vlákien spracúvajúcich detekciu kolízií.

Absolútne maximum so zachovaním plynulosti simulácie[2] bolo dosiahnuté v deme č. 6 so 40 guľami a dĺžkou priemerného časového kroku 27 ms.

[2]Dĺžka časového kroku je kratšia ako 30 ms.

5.2 Diskusia

Behom implementácie som dospel k niekoľkým vlastným záverom. Čo ma prekvapilo asi najviac, je zložitosť fyzikálnej stránky simulácie, a to hlavne vplyv trenia na pohyb objektov. Nájdenie správneho spôsobu vyžadovalo vyskúšanie niekoľkých prístupov a mnohých konzultácií. Taktiež korektné spracovanie kľudových síl vyustilo v pomerne rozsiahly problém a zložitý algoritmus ich riešenia.

Pri testovaní aplikácie som odhalil jednu slabinu impulzového prístupu ku kolíziám, a to predpoklad nulového pôsobenia externých síl počas kolízie. To má v prípade kotúľajúcich sa gúľ po podložke a ich následného nárazu do bočnej steny za následok pomerne veľký skok do výšky, teda proti gravitácii. Tento efekt je pri vhodných nastaveniach simulácie minimálny, ale pri dlhšej práci s enginom viditeľný.

Účinnost 2D mriežky sa podľa nameraných hodnôt v predchádzajúcej sekcii môže zdať dosť malá, avšak je potrebné sa zamyslieť nad typom testovacích scén obsahnutých v demách. Niekoľko z nich predstavuje takmer najhoršie možné prípady[3]. Avšak ak je scéna prirodzená a objekty sú po scéne rozdelené aspoň trochu rovnomerne, tak je schopná znížiť počet porovnávaní až o 50%, čo je veľmi dobrá hodnota.

Veľmi dobré výsledky vykazovalo použitie paralelného prístupu. Už oddelenie prezentačnej vrstvy od výpočtovej vyustilo v priemere takmer o 50% zrýchlenie celého enginu. Paralelizácia detekcie kolízií toto zrýchlenie ešte zvýšila. Na procesore s ôsmymi jadrami sa podarilo v niektorých prípadoch simuláciu zrýchliť až desaťnásobne, čo už je nezanedbateľná hodnota.

[3]Viď demo č. 2, v ktorom sú všetky gule postavené na seba, čím spadajú do rovnakej bunky, a tým je optimalizácia nulová.

6. Záver

V prvej kapitole sme sa zoznámili s problematikou a motiváciou fyzikálne založených simulácií. Následne sme stanovili ciele tejto práce. V druhej kapitole sme sa už sústredili na matematické základy potrebné pre fyzikálne modelovanie vlastností tuhých telies. Začali sme jednoduchými objektami vo forme častíc, ktoré sme následne rozšírili na kompletné tuhé telesá. V oboch prípadoch sa práca zameriavala na popis ich najdôležitejších vlastností a ich vhodnú reprezentáciu. Tretia kapitola sa sústreďovala na problematiku detekcie kolízií a spôsoby jej riešenia. Bol navrhnutý algoritmus a predstavených niekoľko vhodných optimalizácií. Zvyšok kapitoly sa venoval samotným kolíziám, ich deleniu a spracovávaniu. Vo štvrtej kapitole sa zhrnuli ciele implementácie a potrebné technológie pre jej splnenie. Po nich nasledoval bližší popis implementácie dôležitých častí a v prílohách sa nachádza úplna programátorská a užívateľská dokumentácia.

Čo sa týka ďalšieho vývoja, mal by sa uberať podporou kolízií väčšieho množstva objektov. Čo znamená podporu kontaktu, ktorý nie je tvorený len jedným bodom ale celými množinami, napr. hranami alebo celými stenami. Výsledky dokazujú, že paralelizmus má v tejto problematike veľmi dobré uplatnenie, a ešte zďaleka nie je využitý jeho plný potenciál. Ostáva ešte velké množstvo miest, kde by paralelizácia dokázala zvýšiť rýchlosť celej simulácie.

Kvôli praktickej využiteľnosti by bolo vhodné naimplementovať vytváranie objektov mimo zdrojového kódu a poskytnúť možnosť načítavania celých scén z externých súborov.

Literatúra

[1] *Witkin, A., Baraff, D.:* **Physically Based Modeling: Principles and Practice - Differential Equation Basics**
Robotics Institute, Carnegie Mellon University 1997

[2] *Witkin, A.:* **Physically Based Modeling: Principles and Practice - Particle System Dynamics**
Robotics Institute, Carnegie Mellon University 1997

[3] *Baraff, D.:* **An Introduction to Physically Based Modeling: Rigid Body Simulation I. - Unconstrained RigidBody Dynamics**
Robotics Institute, Carnegie Mellon University 1997

[4] *Conger, D.:* **Physics Modeling for Game Programmers**
Robotics Course Technology PTR; 001 edition 2004

[5] *Reinhard, E., Smits B., Hansen, Ch.:* **Dynamic Acceleration Structures for Interactive Ray Tracing**
University of Utah, 2000

[6] *Blow, J.:* **Practical Collision Detection**
Proceedings of the Computer Game Developer's Conference, 1997

[7] *Baraff, D.:* **An Introduction to Physically Based Modeling: Rigid Body Simulation II. - Nonpenetration Constraints**
Robotics Institute, Carnegie Mellon University 1997

[8] *Baraff, D.:* **Fast Contact Force Computation for Nonpenetrating Rigid Bodies**
SIGGRAPH 1994

[9] *Kawachi, K., Suzuki, H, Kimura, F:* **Simulation of Rigid Body Motion with Impulsive Friction Force**
Department of Precision Machinery Engineering, The University of Tokyo, 1997

[10] *Bourg, D:* **Physics for Game Developers**
O'Reilly, 2001

[11] *Wolfgang, Ch., Novak, G.:* **Web Physics - Rolling Friction**
`http://webphysics.davidson.edu/faculty/dmb/py430/friction/rolling.html`, 1995

A. Obsah CD

Súčasťou tejto práce je priložený disk CD-ROM, ktorý obsahuje text práce, zdrojové a spustiteľné súbory. Adresárová štruktúra je nasledovná:

- documents - obsahuje text práce vo formátoch ps a pdf
- executable - obsahuje spustiteľné súbory pre Windows
- project - obsahuje zdrojové súbory, projektové súbory, knižnice a všetky ostatné súbory potrebné pre spustenie a kompiláciu projektu

B. Užívateľská dokumentácia

K dispozícií sú spustiteľné súbory pre Windows, Linuxoví užívatelia si musia aplikáciu skompilovať priloženým *makefile*-om[1]. Informácie o tom kde sa tieto súbory nachádzajú je možné nájsť v prílohe A. Obsah CD.

B.1 Aplikácia

Spustiteľné súbory predstavujú spolu 7 dém, ktoré majú prezentovať fungovanie enginu. Táto aplikácia požaduje minimálne 2 vstupné parametre, kde prvý predstavuje počet vlákien použitých pre spracovávanie kolízii a druhý identifikačné číslo dema. Tieto identifikačné čísla môžu byť vybrané od 0-6. Pri deme čislo 6 je možné 3. voliteľným parametrom definovať počet vygenerovaných gúľ do scény. Príklad volania na OS Windows teda vyzerá napr.

```
CoolPool.exe 2 1
```

Tento príkaz spustí demo číslo 1 a vytvorí 2 vlákna na spracovávanie kolízií. Pri nesprávnom vstupe sa zobrazí chybová hláška s nápovedou. Pre používateľov Windows budú priložené aj .bat súbory pre jednotlivé demá s predefinovaným vstupom, aby nebolo nutné demá púštať cez príkazový riadok.

Pred ukončením programu sa do štandardného výtupu vypíše priemerná dĺžka trvania jedného kroku simulácie, čím je možné sledovať účinnosť paralelizácie.

B.2 GUI

V každom z dém je možné otáčať scénu kliknutím ľavého tlačidla myši a následným pohybom do strán. Taktiež je k dispozícii priblíženie jednoduchým scrollovaním.

Demo s identifikačným čislom 0 obsahuje aj možnosť aktívneho zasahovania do simulácie. To je uskutočnené jednoduchým tágom, ktorým je možne štuchnúť do jednej z gúľ vybraným smerom a silou (viď obrázok nižšie).

Obr. B.1: Jednoduché tágo pre zasahovanie do simulácie, ovládané myšou.

[1]Presný popis je možné nájsť v sekcii C.2.2

C. Programátorská dokumentácia

Zdrojový kód celej aplikácie bol napísaný v jazyku C++ a len za použitia knižníc uvedených v nasledujúcej sekcii.

C.1 Použité knižnice

Pri implentácii bolo použitých niekoľko knižníc:

FreeGLUT - Voľne šíriteľná verzia knižnice GLUT[1]. Ide o prenositeľnú nadstavbu knižnice OpenGL obsahujúcu vytváranie okien, spracovávanie vstupu, vykresľovanie základných primitív a pod.

DevIL - Voľne šíriteľná knižnica, pre prácu s obrázkami a textúrami, táto knižnica je potrebná pre správne fungovanie knižnice FreeGLUT. V zdrojovom kóde nie je priamo využitá.

Boost - Voľne šíriteľná a prenositeľná knižnica, ktorá obsahuje mimo iného hlavne prácu s vláknami v C++, ktorá bola využitá v tomto projekte.

Eigen - Voľne šíriteľná a prenositeľná knižnica lineárnej algebry, podporujúca mimo iného prácu s maticami, vektormi a riešenie sústav rovníc.

FreeImage - Voľne šíriteľná a prenositeľná knižnica pre prácu s obrázkami v rozličných formátoch, použítá na vytváranie textúr.

C.2 Kompilácia

V tejto kapitole bude bližšie popísaný spôsob prekladu zdrojových súborov na jednotlivých platformách. Tieto informácie je možné nájsť aj v angličtine v súbore readme, ktorý je priložený k zdrojovým súborom. Umiestnenie zdrojových súborov je možné nájsť v prílohe A.

C.2.1 Windows

Súčasťou balíka so zdrojovými súbormi je správne nakonfigurovaný *CoolPool.vcproj* súbor pre Microsoft Visual Studio 2008, v prípade používania verzie 2010 je možné tento projektový súbor automaticky skonvertovať.

Pre bezproblémovú kompiláciu je potrebné mať nainštalovanú len knižnicu Boost[2], všetky ostatné knižnice sú priložené.

[1] OpenGL Utility Toolkit

[2] Inštalátor k tejto knižnici je možné nájsť na `http://www.boostpro.com/download/`

C.2.2 Linux

Súčasťou balíka so zdrojovými súbormi je *Makefile*, ktorý umožní kompiláciu prekladačom GCC.

Pred kompiláciou je potrebné si nainštalovať všetky knižnice okrem Eigen-u. Všetky knižnice sú bežne používané a mali by sa nachádzať v repositároch väčšiny distribúcií.

Názvy balíkov pre distribúciu Ubuntu:

- libboost1.42-dev
- libdevil-dev
- libfreeimage-dev
- freeglut3-dev

V prípade inštalácie na inú lokáciu ako */usr/lib* je potrebné pridať dané cesty do priloženého *Makefile*-u.

C.3 Architektúra

V tejto sekcii bude popísaná architektúra enginu, bude obsahovať popis významu a účelu jednotlivých tried a ich metód. Bude taktiež obsahovať jednoduchý návod a popis požiadavkov pre implementáciu nového objektu, ktorý bude môcť byť následne pridaný do enginu.

C.3.1 Jadro

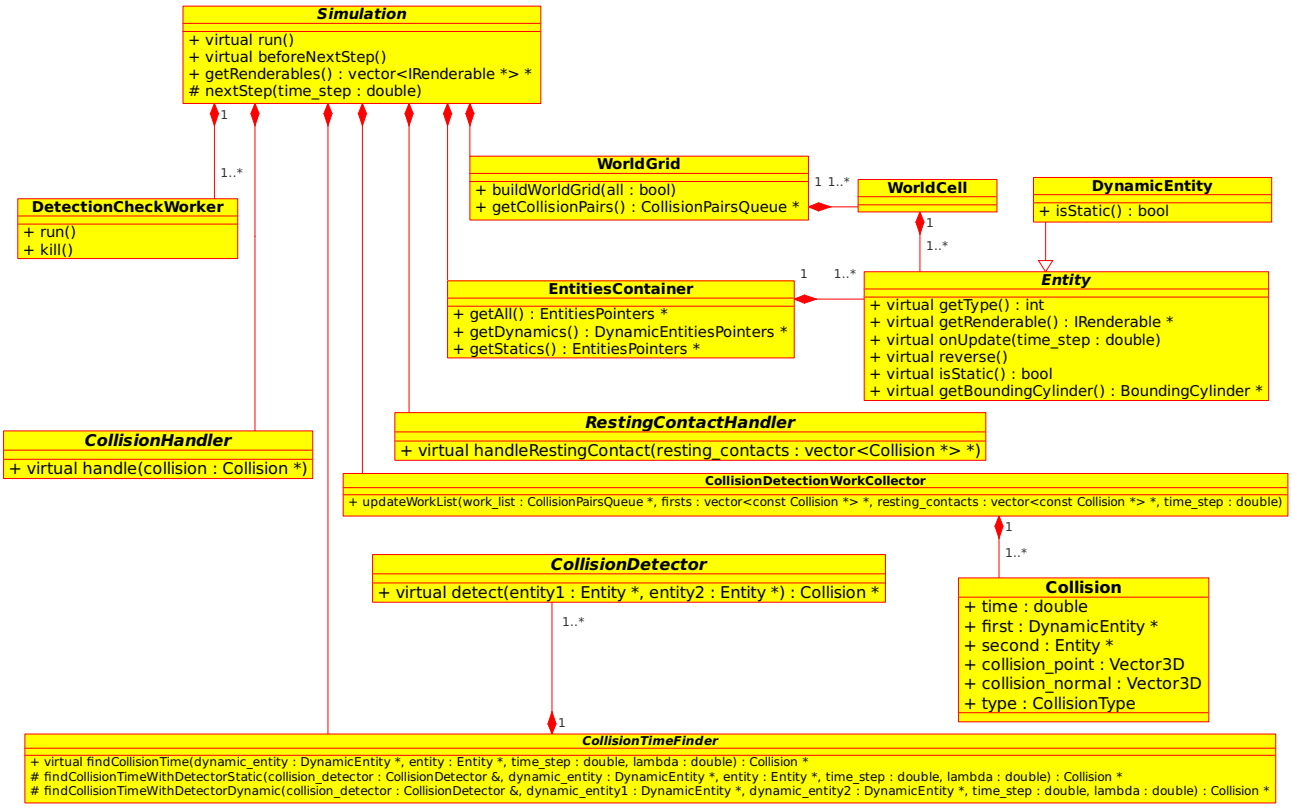
V tejto sekcii bude popísané jadro aplikácie, ktoré je tvorené abstraktnou vrstvou niekoľkých tried. Vzťah jednotlivých tried vo forme UML diagramu tried je možné nájsť na obrázku (C.1) nižšie.

Jadro aplikácie je tvorené abstraktnou triedou `Simulation`, ktorá obsahuje celú hlavnú funkcionalitu simulácie ako aktualizáciu polôh a rýchlostí objektov, hľadanie a spracovávanie kolízií, správa vlákien a všetkých objektov, hlavný cyklus a výstup pre prezentačnú vrstvu. Jej najdôležitejšie metódy sú:

- `void run()` - tvorí štartovací bod pre spustenie v novom vlákne[3], obsahuje hlavný cyklus simulácie a po spustení simulácie je možné ju zastaviť jedine metódou `void kill()`.
- `void beforeNextStep()` - virtuálna metóda, ktorá je volaná pred začiatkom každého kroku simulácie. Každá simulácia môže predefinovať túto metódu podľa potreby, je určená predovšetkým na pridávanie síl ako napr. gravitácia do scény.
- `std::vector<IRenderable *> * getRenderables()`, vráti zoznám všetkých objektov typu `IRenderable`, ktoré sú určene pre prezentačnú vrstvu. Táto metóda nie je synchronizovaná, pred volaním je nutné zamknúť celý objekt pomocou zámku, ktorý je prístupný cez metódu `boost::mutex * getLock()`

[3]Týmto spôsobom sa dosiahne rozdelenie vykresľovania a výpočtov do dvoch vlákien.

- `void nextStep(double time_step)` - metóda, ktorá posunie simuláciu o daný časovy úsek dopredu. Táto metóda je volaná v každej iterácii hlavného cyklu metódy `run()`



Obr. C.1: Zjednodušený UML diagram jadra enginu.

Entity je abstraktná trieda, ktorá tvorí základ všetkých objektov simulácie. Obsahuje prevažne virtuálne metódy cez ktoré simulácia komunikuje s objektami. Všetky objekty musia tieto metódy korektne prepísať, ak chcú byť súčastou simulácie. Tieto metódy sú:

- `int getType()` - vráti typ objektu v tvare celého čísla, toto typovanie je statické a je úplne ponechané na programátora, je možné ho využiť napríklad v miestach spracovania kolízii a podobne, kde je potrebné rozlišovať medzi typmi objektov. Simulácia samotná sa nijak o typy objektov nestará.

- `IRenderable * getRenderable()` - vráti objekt, ktorý je predaný prezentačnej vrstve a obsahuje všetky informácie potrebné na jeho vykreslenie. Tento objekt môže byť ľubovoľný a závisí len od používateľa ako a čím sa objekt vykreslí.

- `void onUpdate(double time_step)` - táto metóda je volaná na objekte v každom kroku simulácie, vo väčšine prípadov obsahuje posun predmetu na základe jeho rýchlosti, ale v prípade potreby môže byť použitý aj na jednoduché animácie a pod.

- `void reverse()` - vráti objekt do stavu pred posledným `onUpdate(double time_step)`, táto metóda je používaná detektorom kolízií, pre hľadanie presného času kolízie.
- `bool isStatic()` - vráti `true` v prípade ak ide o statický objekt a `false` v opačnom prípade. Tento fakt je použitý na optimalizáciu detekovania kolízií.
- `BoundingCylinder * getBoundingCylinder()` - vráti `BoundingCylinder` daného objektu, ten je použitý pre rýchle vkladanie objektu do 2D mriežky[4].

DynamicEntity je potomkom triedy `Entity` s prepísanou metódou

```
bool isStatic()
```

tak, že vracia `false`.

Collision je v skutočnosti štruktúra, ktorá slúži na zber všetkých potrebných informácii o kolízii. Na základe týchto dát je kolízia následne ošetrená. Jej atributy sú:

- `double time` - čas kolízie v rámci časového intervalu
- `DynamicEntity * first` - prvý objekt kolízie, ten musí byť určite dynamický
- `Entity * second` - druhý objekt kolízie môže, ale nemusí byť dynamický
- `Vector3D collision_point`[5]- miesto kolízie v svetových súradniciach
- `Vector3D collision_normal` - vektor kolmý na rovinu kolízie
- `CollisionType type` - typ kolízie, môže nadobudnúť hodnoty:
  - `MOVINGAWAY` - typ kolízie, kedy sa objekty od seba vzďalujú a nie je potrebné ju riešiť
  - `INTERPENETRATION` - typ klasickej kolízie, s nárazom
  - `RESTING_CONTACT` - objekty sa dotýkajú a sú v tzv. pokojovom kontakte

CollisionDetector je abstraktná trieda, ktorej metóda

```
Collision * detect(Entity * entity1, Entity * entity2)
```

musí vrátiť ukazateľ na objekt `Collision` v prípade ak dané dva objekty kolidujú, teda sú v prieniku. V opačnom prípade musí vrátiť `NULL`. Táto metóda musí nastaviť všetky atributy kolízie okrem času, tá je doplnená inštanciou triedy `CollisionTimeFinder`. O správne využitie a vymazanie objektu `Collision` sa postará simulácia. Táto metóda určite potrebuje rozlišovať medzi typmi objektov, a preto by mala mať každá dvojica objektov jeden `CollisionDetector`. O správny vstup do metódy sa musí starať znovu inštancia triedy `CollisionTimeFinder`. Táto metóda musí byť synchronizovaná.

[4]Bližší popis oboch objektov je možné nájsť v nasledujúcom texte.
[5]Vector3D predstavuje klasický trojzložkový vektor.

CollisionTimeFinder je abstraktná trieda, ktorej metóda

```
void findCollisionTime(DynamicEntity * dynamic_entity, Entity * entity,
double time_step, double lambda)
```

má za úlohu nájsť presný čas kolízie medzi danými dvoma objektami. V skutočnosti však jej úlohou je len volanie jednej z metód

```
Collision * findCollisionTimeWithDetectorStatic(CollisionDetector &
collision_detector, DynamicEntity * dynamic_entity, Entity * entity,
 double time_step, double lambda)
```

```
Collision * findCollisionTimeWithDetectorDynamic(CollisionDetector &
collision_detector, DynamicEntity * dynamic_entity1,
DynamicEntity * dynamic_entity2, double time_step, double lambda)
```

so správnymi vstupnými hodnotami, teda správna dvojica objektov pre daný `CollisionDetector`. Ďalší vstupný parameter `double time_step` určuje časový krok v ktorom je kolízia hladaná. Parameter `double lambda` je len možná optimalizácia a obsahuje čas, po ktorom kolízia už nie je hľadaná. To z dôvodu, že kolízie sa spracúvajú postupne od tej prvej po poslednú, teda `lambda` obsahuje najskoršiu kolíziu, ktorá už bola nájdená. Tieto 2 metódy robia v podstate to isté, zas ide len o optimalizáciu pre statické objekty, pretože v prípade statického objektu nemá zmysel ho posúvať dopredu a zas vracať do predošlého stavu. Prístup k tomuto objektu musí byť znovu synchronizovaný.

CollisionHandler je abstraktná trieda, ktorá obsahuje len jednu virtuálnu metódu a to

```
void handle(Collision * collision)
```

Táto metóda musí byť schopná ošetriť kolíziu typu `INTERPENETRATION` daných dvoch objektov na základe informácií, ktoré je možné nájsť v štruktúre `Collision`. Vo väčšine prípadov je na tejto úrovni potrebné objekty rozlišovať, na to je možné použiť vyššie spomínanú metódu `getType()` definovanú na každom objekte. Toto rozlišovanie v tom prípade musí byť súčasťou tejto metódy.

RestingContactHandler je abstraktná trieda, ktorá ma za úlohu riešiť špeciálny typ kolízie a to `RESTING_CONTACT`. K tomu obsahuje metódu

```
void handleRestingContact(std::vector<const Collision *> *
resting_contacts)
```

ktorej prvým argumentom je zoznam všetkých týchto kolízií za jeden krok simulácie. Všetky tieto kolízie musia byť spracované naraz. Táto metóda nemusí byť synchronizovaná.

EntitiesContainer je trieda, ktorej úlohou je spravovať a poskytovať prístup k všetkým objektom simulácie. Poskytuje 3 základné metódy:

```
EntitiesPointers * getAll();

DynamicEntitiesPointers * getDynamicEntities();

EntitiesPointers * getStaticEntities();
```

ktoré vracajú ukazatele na zoznamy objektov daného typu v konštantnom čase. `EntitiesPointers, DynamicEntitiesPointers` predstavujú typy definované ako

```
typedef std::vector<Entity *> EntitiesPointers;
typedef std::vector<DynamicEntity *> DynamicEntitiesPointers;
```

Treba podotknúť, že tento kontajner predpokladá, že zoznam objektov a ich typy vzhľadom k statickosti resp. dynamickosti sa nemenia počas simulácie. Tieto zoznamy sa nainicializujú pri vytvorení simulácie a sú ďalej nemenné. Prístup k tomuto objektu musí byť synchronizovaný.

CollisionDetectionWorkCollector je trieda, ktorá zaručuje paralelné spracovávanie detekcie kolízií. Jej najdôležitejšou metódou je

```
void updateWorkList(CollisionPairsQueue * work_list,
std::vector<const Collision*> * firsts,
std::vector<const Collision *> * resting_contact, double time_step)
```

ktorej jednotlivé vstupné parametre predstavujú:

- `CollisionPairsQueue * work_list` - fronta dvojíc objektov, ktoré je potrebné skontrolovať na kolíziu definovaná ako
  `typedef std::queue<CollisionPair> CollisionPairsQueue;`
- `std::vector<const Collision*> * firsts` - zoznam, do ktorého sa zapíšu najskoršie kolízie typu `INTERPENETRATION` za daný časový úsek[6]
- `std::vector<const Collision *> * resting_contact` zoznam, do ktorého sa zapíšu všetky kolízie typu `RESTING_CONTACT` za daný časový úsek
- `double time_step` - časový úsek, v ktorom sa majú kolízie hľadať

Po každom zavolaní tejto metódy, sa pošle signál všetkým vláknam, aby začali spracovávať tieto dvojice objektov a testovať ich na kolíziu. Tieto vlákna sa uspia hneď potom, ako je zoznam dvojíc na spracovanie prázdny a ony skončili všetky výpočty. Prístup k jednotlivým vstupným zoznamom je synchronizovaný. Simulácia je nútená počkať, kým sa tieto dvojice nespracujú a zoznamy nenaplnia.

[6]Pri veľkom, množstve objektov sa môže stať, že nastane viac kolízií naraz, aj napriek zmenšeniu časového kroku, v takom prípade sú spracované naraz.

DetectionCheckWorker je trieda reprezentujúca jedno vlákno paralelného spracovávania detekcie kolízií. Toto vlákno sa vytvorí na začiatku simulácie s odkazom na inštanciu triedy `CollisionDetectionWorkCollector` a potom ako je spustené v novom vlákne pomocou metódy

```
void run()
```

spracuváva kolízie až do jeho ukončenia metódov

```
void kill()
```

Toto vlákno sa uspí vždy hneď potom, ako sú spracované všetky dvojice a pred každou dávkou nových dvojíc je potrebné ho zobudiť.

BoundingCylinder je trieda predstavujúca najmenší valec, ktorý svojou veľkosťou úplne obsiahne zvolený objekt. Os tohto valca je rovnobežná s osou y, a preto na jeho definíciu stačia súradnice stredu, ktoré sú dané 2-vektorom a jeho polomerom.

WorldGrid je trieda predstavujúca 2D mriežku popísanú v kapitole (3.1.2). Jej stred je kvôli zjednodušeniu vkladania v strede súradnicovej sústavy a rozlieha sa v smere kladnej x-ovej a z-ovej osy. Objekty mimo tejto hranice sú ignorované, preto je potrebné pred vytvorením scény posunúť objekty do tejto oblasti súradnicovej sústavy. Veľkosť tejto mriežky je naalakovaná pri jej vytvorení tak, aby pokryla všetky objekty simulácie v aktuálnom stave. Ak sa objekty nejakým spôsobom dostanú mimo tejto zóny, budú opäť ignorované. Veľkosť bunky je vytvorená na základe polomeru `BoundingCylinder`-u najväčšieho dynamického objektu, ktorý je následne vynásobený konštantou `WORLDCELL_SIZE_SCALE`. Táto mriežka je tvorená inštanciami triedy `WorldCell` a pre zaradenie objektu do vhodnej bunky používa jeho `BoundingCylinder` popísaný vyššie. Trieda má priamy prístup k inštancii triedy `EntitiesContainer`, ktorá je jej predaná pri vytváraní a vytvára mriežku nad objektami nachádzajúcimi sa v tomto kontajneri. Jej najdôležitejšie metódy sú

```
void buildWorldGrid(bool all)
```

```
CollisionPairsQueue * getCollisionPairs()
```

pričom prvá z nich aktualizuje mriežku podľa aktuálnych pozícii objektov. Jej parameter rozhoduje, či bude aktualizovať mriežku zo všetkých objektov alebo len statických. Obvykle stačí túto metódu zavolať s parametrom `true` len raz na začiatku a potom aktualizovať len dynamické objekty. Druhá metóda vráti všetky páry objektov, ktoré treba otestovať na kolíziu, tie získa postupným prechodom všetkých buniek mriežky.

WorldCell je trieda, ktorej inštancia predstavuje jednu bunku 2D mriežky tvorenej triedou `WorldGrid`. Táto bunka obsahuje zoznam všetkých objektov, ktoré ju pretínajú.

IRenderable je abstraktná trieda obsahujúca jedinú virtuálnu metódu a to

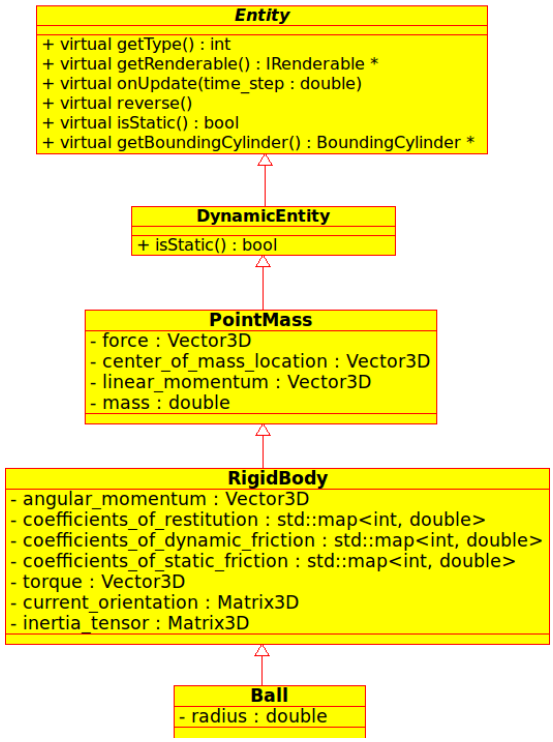
`void render(),`

ktorá ma za úlohy vykresliť nejaký objekt, nie je špecifikované čím alebo kým, táto voľba ostáva na programátorovi používajúcom tento engine. Každý objekt simulácie je povinný vytvoriť takýto objekt po zavolaní metódy `getRenderable()`. Čiže pri zmene prezentačnej vrstvy stačí prepísať jedinú metódu vytvárajúcu tento objekt.

C.3.2 Objekty

V kapitole 2 bol predstavený matematický model vyjadrujúci stavbu objektov. V tejto sekcii bude bližšie popísaná ich objektová reprezentácia a implementácia. Kvôli väčšej prehľadnosti je k dispizícii UML diagram tried, ktorý je možné nájsť nižšie.

Entity, DynamicEntity predstavuje základ objektov, ktorý je podrobnejšie popísaný v predošlej kapitole. Všetky objekty simulácie musia vychádzať z nich a správne implementovať všetky potrebné metódy, cez ktoré s nimi simulácia bude komunikovať.



Obr. C.2: Zjednodušený UML diagram tried reprezentujúci hierarchickú stavbu jedného objektu simulácie.

PointMass je pre zhrnutie trieda reprezentujúca objekt vyjadrený len svojou polohou a váhou. Jeho atributy preto sú

- `Vector3D center_of_mass_location` - poloha objektu v priestore

- `Vector3D linear_momentum` - hybnosť objektu[7]
- `Vector3D force` - sila, ktorá pôsobí na teleso za jeden časový krok simulácie, tento vektor je vynulovaný po každom kroku simulácie
- `double mass` - hmotnosť objektu

Prístup k jednotlivým atributom je uskutočnený príslušnou `get` metódou a nastavenie zas metódou `set`.

RigidBody je trieda, ktorá je potomkom triedy `PointMass` a obsahuje prevažne reprezentáciu rotačnej zložky pohybu objektu. Jej atributy preto sú

- `Matrix3D current_orientation` - matica rotácie vyjadrujúca aktuálnu orientáciu telesa v priestore
- `Vector3D angular_momentum` - uhlový moment telesa
- `std::map<int, double> coefficients_of_restitution` - koeficienty pružnosti, pár (typ, koeficient) pre každú dvojicu v prípade, ak nie je definovaná simulácia počíta s koeficientom 1
- `std::map<int, double> coefficients_of_dynamic_friction` - koeficienty dynamického trenia pre jednotlivé objekty definované, ako v prípade koeficientu pružnosti
- `std::map<int, double> coefficients_of_static_friction` - koeficienty statického trenia pre jednotlivé objekty definované, ako v prípade koeficientu pružnosti
- `Vector3D torque` - krútiaci moment pôsobiaci na teleso behom jedného časového kroku simulácie, tento vektor je vynulovaný po každom kroku simulácie
- `double inertia_tensor` - matica tensoru uhlového momentu

Prístup k jednotlivým atributom je uskutočnený prúslušnou `get` metódou a nastavenie zas metódou `set`.

Ďalšia úroveň už predstavuje implementáciu konkrétnych tuhých telies. Pre úplnosť ešte popíšem implementáciu gule.

Ball je trieda, ktorá je priamym potomkom `RigidBody`, je to prakticky najjednoduchšie tuhé teleso. Predpokladáme, že jeho hmotnosť je rovnako rozložená po celom jeho objeme. Jeho tensor uhlového momentu je tak veľmi jednoduchý a jeho vynásobenie má len efekt vynásobenia každej zložky vektoru rovnakým skalárom. Líši sa od ostatných objektov len tým, že implementuje ešte válivé trenie, ktorého koeficienty pre jednotlivé objekty sú uložené v atribute

`std::map<int, double> coefficients_of_rolling_friction`

[7]Bola vybraná miesto rýchlosti kvôli konzistencii s ostatnými objektami.

C.3.3 Pridanie nového telesa

V tejto sekcii bude popísaný návod ako pridať nový objekt do simulácie. Tento objekt bude pre jednoduchosť statický, a to z dôvodu, že pridanie dynamického objektu by mohlo vyžadovať nejaké zmeny navyše, pretože súčasný engine počíta s kolizným kontaktom tvoreným len jedným bodom.

Ako príklad nám poslúži valec, ktorý by sme chceli pridať do simulácie. Vytvorme si konštantu predstavujúcu typ tohto objektu a nazvime ju `CP_CYLINDER`. Môžme jej priradiť ľubovoľnú hodnotu, ale nesmie sa zhodovať s typom už definovaného telesa. Tieto konštanty sú definované v súbore `ObjectTypes.h`. Trieda reprezentujúca toto teleso musí byť potomkom triedy `RigidBody`, v nej je možné definovať reprezentáciu tohto telesa.

Je potrebné predefinovať všetky metódy, ktoré su vyžadované od prapredka a teda triedy `Entity`. Metódu `getType()`, predefinujeme tak aby vracala práve definovanú konštantu `CP_CYLINDER`. Ďalej je potrebné predefinovať metódu `isStatic()` tak, aby vracala `true`, keďže pôjde o statický objekt. Metódy `onUpdate(double time_step), reverse()` sa kvôli jednoduchosti predefinujú na prázdne metódy, tým nebudú nijak zasahovať do behu simulácie. Ďalšiu metódu, ktorú je potrebné predefinovať je metóda `getRenderable()`, táto metóda musí vracať ukazateľ na potomka triedy `IRenderable`, v nej je potrebné definovať vykresľovanie valca. Dobrým príkladom podobnej triedy, ktorým je možné sa nechať inšpirovať, je trieda `BallRenderable`, ktorej metóda `render()` vyzerá nasledovne

```
void BallRenderable::render() const
{
        GLUquadric * cylinder_obj_ = gluNewQuadric();
        gluQuadricTexture(cylinder_obj_, GL_TRUE);
        glColor3d(1.0, 1.0, 1.0);
        glEnable(GL_TEXTURE_2D);
        glBindTexture(GL_TEXTURE_2D, texture_);

        glPushMatrix();
        glTranslated(location_.getX(), location_.getY(), location_.getZ());
        glMultMatrixd(orientation_);
        gluSphere(cylinder_obj_, radius_, 30, 30);
        glPopMatrix();
        gluDeleteQuadric(cylinder_obj_);
}
```

pričom informácie o lokácii a polomere sú jej predané pri vytváraní. Ďalej je potrebné prepísať metódu `getBoundingCylinder()`, ktorá má vrátiť najmenší valec rovnobežný s osou y, ktorý obsiahne celý objekt. Keďže ide o statický objekt, tento valec stačí vytvoriť len raz pri vytvárani celého telesa.

Poslednou vecou, ktorú je potrebné definovať je detektor kolízií. Teda potomka triedy `CollisionDetector`, ktorý bude schopný rozhodnúť o kolízií valca s ostatnými objektami. V súčasnom prípade, kedy sa v scéne nachádzajú ako dynamické objekty len gule, stačí definovať jediný detektor pre dvojicu valec-guľa. Tento detektor musí vytvoriť validný objekt typu `Collision`, do ktorého musí

zaznamenať všetky informácie o kolízii. Bližší popis tohto objektu a jeho atributov je možné nájsť v sekcii (C.2.1). Takto vytvorený detektor kolízií je ešte potrebné pridať do triedy, ktorá je potomkom `CollisionTimeFinder`. V súčasnosti používaný potomok tejto triedy sa nazýva `BallCollisionTimeFinder`, ktorého metóda `findCollisionTime(...)` vyzerá nasledovne

```
const Collision * findCollisionTime(DynamicEntity * dynamic_entity,
Entity * entity,  double time_step, double lambda)
throw (UnexpectedInputException)
{
        const Collision * output = NULL; // No collision detected.
        switch (entity->getType())
        {
                case CP_BALL:
                {
                        output = findCollisionTimeWithDetectorDynamic(
                        bb_detector, dynamic_entity,
                        static_cast<DynamicEntity*> (entity),
                         time_step,
                        lambda);

                        break;
                }
                case CP_RECTANGLE:
                {
                        output = findCollisionTimeWithDetectorStatic(
                        br_detector,
                        dynamic_entity,
                        entity,
                        time_step,
                        lambda);

                        break;
                }
                default:
                // Uknown objects are ignored
                        break;
                }
        return output;
}
```

Pre vloženie nového detektoru stačí pridať novú cestu pre typ `CP_CYLINDER` a zavolať metódu `findCollisionTimeWithDetectorStatic` na novo vytvorený detektor kolízií.

Ak boli všetky metódy korektne predefinované, tak inštancie tohto objektu je možné pridávať do scény a očakáva sa ich korektné správanie.